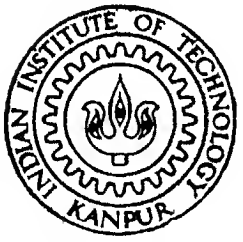
Lightweight Communication Protocol for Distributed Computing

by

Ritvik Ranjanam Pandey



Department of Electrical Engineering
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
April, 1998

Lightweight Communication Protocol for Distributed Computing

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Ritvik Ranjanam Pandey

to the

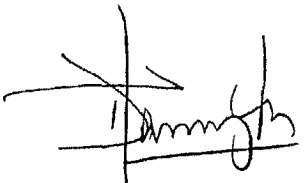
DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

April 1998

Certificate

This is to certify that the work contained in the thesis entitled Lightweight Communication Protocol for Distributed Computing by Ritvik Ranjanam Pandey has been carried out under my supervision and that this work has not been submitted elsewhere for a degree



April 1998

Dr D Manjunath
Assistant Professor
Department of Electrical Engineering,
Indian Institute of Technology Kanpur

Acknowledgments

First, I would like to thank the supernatural power that made this thesis completed by me, against whose will nothing is possible

Next, I would like to thank my guide Dr D Manjunath, for the benevolent guidance and motivation provided by him I would like to thank him for providing me a free atmosphere of work which doubled my motivation His faith and confidence in me acted as a catalyst for my work efficiency His knowledge about my limitations abilities and interests made me work with my full efficiency

I would like to thank Mr K Shyamsundar Mr A Roy Mr P V K Reddy and people in the ERNET lab who were continuously troubled by me to end my troubles without troubling me a little

I would be selfish if I do not thank my friends who spoiled my precious time for spoiling it and to those who did not spoil my time for not spoiling it

Thanks to my family members for the invaluable support and the continuous encouragement provided by them

Nothing would have been possible if the management of various facilities like Library Computer Center etc was not proper Thanks to those who are there in the backdrop to provide a congenial atmosphere

Ritvik Pandey

Abstract

Applications like distributed computing need frequent and intensive transaction of data over a communication network. Schemes like Message Passing Interface (MPI) provide communication libraries, in addition to others, to effect distributed computing over a network. Most implementations of these libraries use TCP/IP protocols for transport and network layer functions while the libraries themselves reside in the application layer. Since TCP/IP is designed to work reliably in very large networks too, it is bound to be slow and inefficient for small, high performance, reliable networks. Due to this, the transport layer becomes the bottleneck in the computing speed a considerable amount of time is spent in communicating.

This thesis designs and implements a lightweight communication protocol, LeghtCommunicator, to substitute the heavyweight TCP/IP stack in distributed computations over small, high speed LANs. The substitute offers the same reliability characteristics as that of TCP/IP but has a processing delay of half that of TCP/IP.

In the design of LightCommunicator, we assume that all communication is over the same LAN.

Contents

List of Figures

Chapter 1

Introduction

To support parallel programming over a network of computers several schemes have been proposed Most of these schemes provide a library of functions to facilitate inter processor communication and constructs for parallel programming to a standard high level programming language Examples of such schemes include the Message Passing Interface (MPI) Parallel Virtual Machine (PVM) Chameleon Chimp Zipcode etc These include basic functions to initiate a session of parallel execution of program sending and receiving messages closing a session etc Apart from these basic functions some advanced functions like for example functions to create separate subgroups functions for process management and error handling functions for collective messaging, etc are also provided These schemes are designed primarily for master slave architectures and support a variety of interconnection topologies Although the performance of the parallel computer formed using these libraries is not as good as that of multiprocessor computers they offer a cheap and useful alternative

At IIT Kanpur MPI has been extensively used for distributed scientific computing It is generally believed by this user community that the effectiveness of a distributed computing setup is not completely evident and we believe that this could be because MPI uses TCP/IP in its communication library This seems reasonable because TCP/IP is a general purpose communication protocol suite meant for bidirectional communication even across hostile networks To improve the communication efficiency of the MPI library especially when the computers are connected over a high performance LAN, we decided to develop LightCommunicator, a lightweight communication protocol stack that interfaces

like the TCP/IP but improves upon its delay performance especially in distributed computing environments using MPI

The MPI standard is a complete interface for message based interprocess communication specified by a multilateral gathering of parallel computing users vendors and researchers The MPI standard provides true portability to parallel programs The message passing paradigm of MPI imparts a distributed memory characteristic to the multicomputer which adds to the portability Most MPI implementations are designed to also work on heterogeneous systems The MPI standard is flexible enough to run on a network of computers as well as on standalone multiprocessor computers MPI provides convenient C and Fortran 77 bindings for interface

The MPI standard does not include implementation issues To have a closer look at the implementation we will consider LAM MPI (Local Area Multicomputer/Message Passing Interface) developed at Ohio Supercomputer Center It is written in two layers The upper layer is portable and independent of the communication subsystem It interfaces to the lower layer through the Request Progression Interface (RPI) consisting of eight functions This interface uses Internet domain TCP sockets as the communication subsystem using the TCP/IP as the underlying protocol

Most MPI implementations like LAM MPI for example, provide a parallel programming environment over a network and use TCP/IP for the transport and network layer protocol for data transaction This makes the implementations portable because TCP/IP is a widely used communication protocol Thus the MPI library functions lie entirely in the application layer The part of the implementation that deals with the communication issues consists mainly of routines which interface the MPI implementation with the TCP/IP stack of the kernel

Since TCP/IP is designed for a wide variety of applications, it may lead to poor end to end throughputs in certain applications In applications like distributed computing with message passing, low communication delay is a necessity This is because in such systems, memory is distributed and to maintain coherence, frequent message passing is necessary Also, computations which need parallel programming typically process large amounts of data and these need to be exchanged during the computation Thus a distributed computing environment incorporates an intensive and frequent message

passing In such situations small inefficiencies in communication can lead to large overall delays In addition to the delays due to the nature of the communication protocol delays can also be increased due to implementation inefficiencies For example the number of copies of a message that are made while it traverses through the stack depends on the implementation and not on the protocol characteristics

In this thesis we present the design and implementation of a lightweight communi cation protocol called LightCommunicator for distributed computing applications over friendly LANs LightCommunicator will provide a reliability to the applications simi lar in capability to that of TCP/IP without offering the overheads of TCP/IP in an intensive, frequent and unidirectional data transaction over a local area networks Light Communicator is implemented inside the Linux kernel The final implementation fits in the kernel like other standard protocols already implemented in the Linux kernel Thus the protocol can coexist in the kernel with other communication protocols LightCom municator as seen by the application layer, is exactly similar to TCP/IP but differs from it in implementation This feature minimises the changes required in the source code of the application layer programs to replace TCP/IP by LightCommunicator

The next chapter discusses various overheads of TCP/IP and why they are unneces sary in distributed computing environments Chapter 3 discusses the implementation of the network module in the Linux kernel It also discusses the mechanism with which the requests for data transaction are processed in the Linux kernel Chapter 4 explains the details of LightCommunicator and its implementation The last chapter presents some experimental results that study the delay performance of LightCommunicator vis a vis TCP/IP It also includes summary of the work and suggests some future developments

Chapter 2

The TCP/IP Protocol Suite

2 1 Introduction

In early 1980's the Advanced Research Projects Agency (ARPA) of the US Department of Defence (DoD) specified a new family of protocols as the standard for the ARPANET sponsored by it This protocol suite, accurately known as DARPA Internet protocol suite' is widely known as the TCP/IP protocol suite or simply TCP/IP (Transport Control Protocol/Internet Protocols)

TCP/IP has many interesting features It is not vendor specific and can be implemented on lowly personal computers as well as on large supercomputers, it can be used for both LANs and WANs Today TCP/IP links millions of nodes around the globe using all possible physical media like telephone lines coaxial cables (Ethernet links), fiber optic cables and satellite channels One of the reasons of the popularity of TCP/IP was its inclusion in the BSD Unix systems very early Subsequently its inclusion in other operating systems like Unix System V added to its popularity

The general nature of TCP/IP is obviously implied by the extensive service it provides to a variety of networks supporting a wide spectrum of applications like HTTP FTP, STMP etc This capability for extensive service makes the TCP/IP protocol suite too sophisticated for small, reliable, high speed networks because most of its features become redundant and eventually become the bottleneck in achieving high end-to end throughput

2 2 Organisation of TCP/IP

The basic skeleton of the TCP/IP is a simplified OSI stack containing only four layers viz application layer transport layer network and the data link layer [3] (See Figure 2 1)

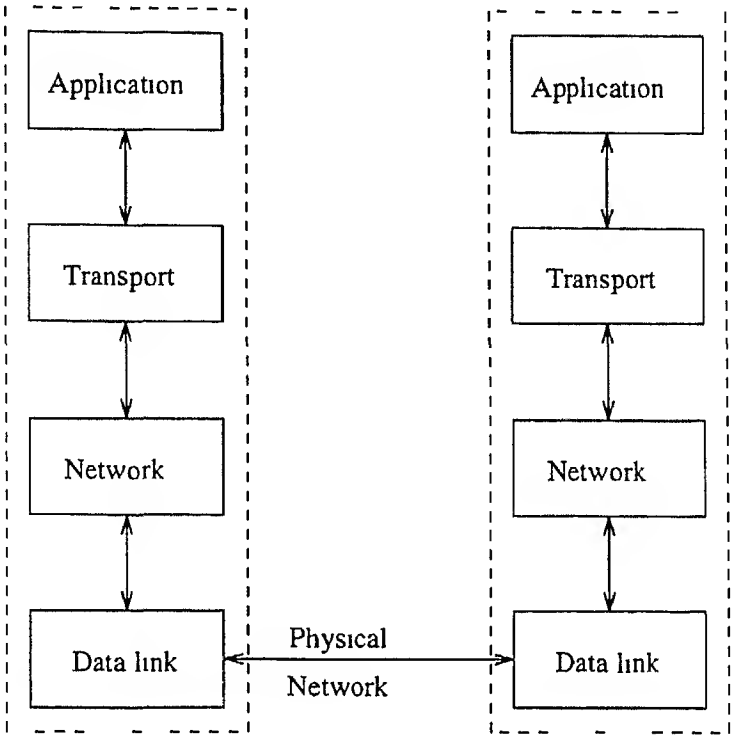


Figure 2 1 Simplified 4 layer model used by TCP/IP

There are many protocols in the TCP/IP suite and comprise a family referred to as the TCP/IP family This family is a collection of many members organised as shown in Figure 2 2 Transmission Control Protocol (TCP) is a connection oriented protocol that provides a reliable full duplex byte stream for a user process It is the most widely used protocol of this family User Datagram Protocol (UDP) is a connectionless protocol for user processes It is unreliable i e there is no guarantee that a datagram sent will reach the destination Internet Control Message Protocol (ICMP) is used to handle error and control information between gateways and hosts This protocol is not meant to serve user processes but carries messages generated by the TCP/IP networking software Internet Group Management Protocol (IGMP) is the part of TCP/IP dealing with the broadcast and multicast functions of the protocol

Internet Protocol (IP) is the network layer serving the transport layer occupied by TCP, UDP, ICMP and IGMP It does not carry the messages produced by user processes

directly Nodes in a TCP/IP network or Internet have an address called the IP address This is used to uniquely identify an Internet connection This address is different from the hardware (data link layer) address used by a node in the network to which it is connected Address Resolution Protocol(ARP) is a protocol to serve the queries of IP about the hardware addresses to be assigned to an outgoing packet Reverse Address Resolution Protocol (RARP) is a protocol used to map hardware addresses to IP addresses

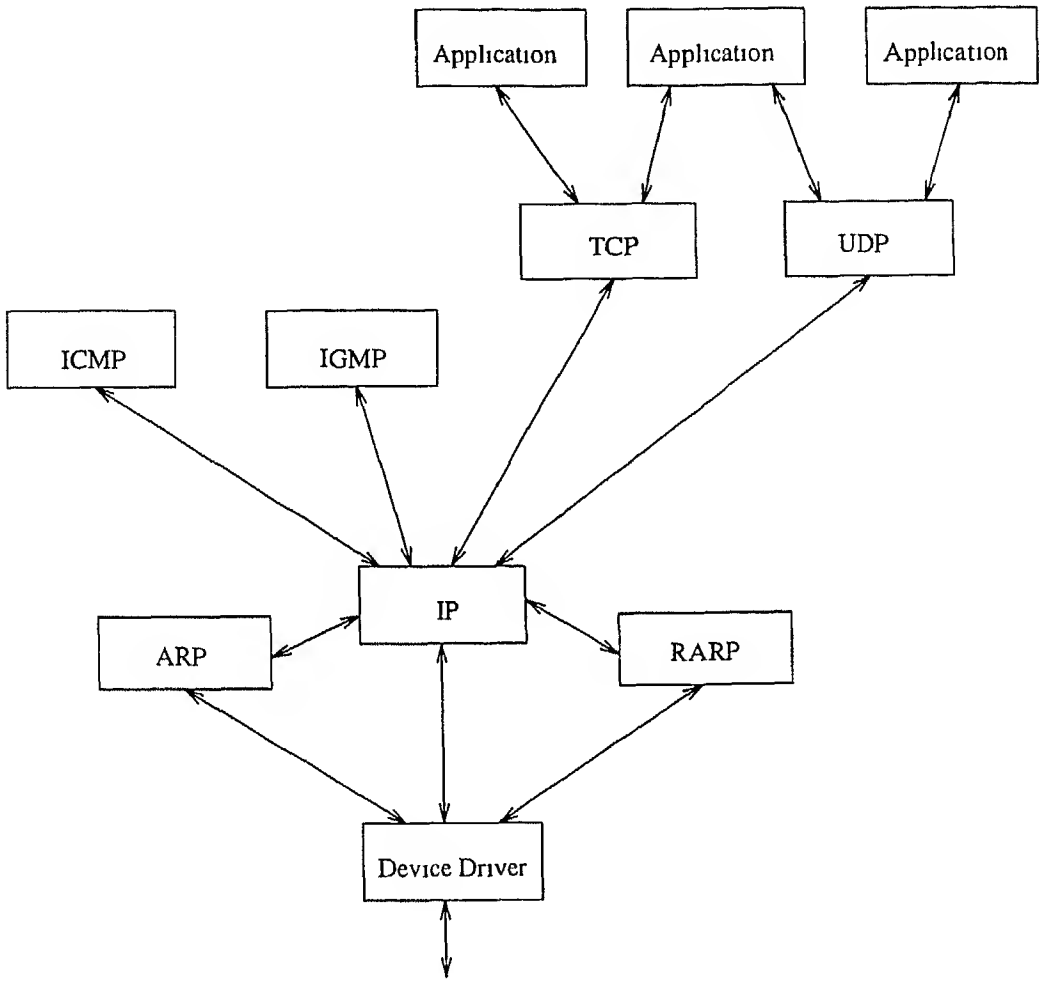


Figure 2 2 TCP/IP protocol family

The above layout follows the four layer model shown in figure 2 1

2 3 The Network Layer - IP

The IP layer provides an unreliable delivery of the data streams supplied by the transport layer The IP layer does not keep any account of the packets sent by it An outgoing

packet is completely independent of the preceding and the succeeding packets No sequencing is done in this layer Reliability of data delivery is provided by the upper layers

For addressing IP uses a 32 bit address, which is unique for each network connection and is organised in four categories viz Class A B C and D addresses The IP address has some bits dedicated to denote the class to which it belongs some to the network and rest to the host Such an organisation is necessary because of the size of the internetwork and the variety of networks served by the protocol An IP packet contains a 20 (and some options if they exist) byte IP header followed by the data to be transmitted

The basic function of the IP layer is to do a translation of the IP addresses to the hardware address to which the packet has to be sent Since hardware addresses are not unique and in more general cases when there are different networks e g , Ethernet Token Ring etc , there might not be any relation between two hardware addresses This mapping may also be dynamic in time ARP and RARP provide the mapping on demand for IP ARP and RARP are full fledged protocols in themselves (This mapping will need to be to be done at every router that a packet passes through on its way to the destination) In a LAN environment, the ARP and ARP processing are not required because the nodes have unique hardware addresses and a host can be uniquely identified by this address

Another important function of IP is *fragmentation* of outgoing packets This is necessiated by the fact that the data link layer might not allow arbitrarily long packets to be transmitted on the network The packet size is limited by the Maximum Transmission Unit (MTU) of the data link layer IP is also equipped to handle situations where a packet moves from a network of higher MTU to a network which that supports a lower MTU This power of IP also is of no use if the packet does not cross a bridge

There are many more functions of IP like forwarding, transparent proxying, masquerading etc which are of no use when a packet moves within a LAN

2 4 The Transport Layer - TCP and UDP

TCP is more complicated than IP The strength of the family is provided by this protocol It performs a wide range of functions It is highly reliable and has powerful mechanisms

to control the flow of data

TCP is a connection oriented protocol i e two applications before using TCP for data transaction establish a connection Establishment of connection is a three way process Unless a connection is established two terminals cannot communicate with each other After the data transaction is complete, they have to terminate the connection A more detailed discussion will be done in § 2 4 1 and § 2 4 2

To ensure a reliable flow of data TCP breaks the application data into small *segments* before delivering it to IP A TCP segment contains a TCP header of 20 bytes (and some option bytes if any) followed by data On the receiver side, TCP sends *acknowledgments* for the data it receives The process of sending acknowledgments is slightly involved and will be discussed in § 2 4 4 A TCP transmitter can send segments whose sequence numbers lie within a *window* starting from the last segment number for which the ac knowledgment was received After sending a segment it starts a timer and if before the expiry of the timer it does not receive an acknowledgment it will retransmit the segment These retransmissions can cause duplicate instances of same segment at the destination which are to be discarded by TCP If segments arrive out of order they have to be rearranged To ensure in order and reliable delivery of data, TCP also maintains a *checksum* of the whole segment

TCP is also responsible for controlling congestion in the network This it does by dynamically changing the window size and the *timer* timeouts Several techniques like *delayed acknowledgments Nagle s rule, piggybacking* etc are used to improve end to end throughput by TCP

A TCP segment is handed over to IP layer, which adds its own header and gives it to the data link layer The data link layer adds its header and trailer and finally transmits it on the network Figure 2 3 shows the encapsulation of data as it goes down the protocol stack

2 4 1 Connection Establishment

TCP uses a three way handshaking in connection establishment procedure First, a request is made from the client side for a connection by sending a SYN signal then the server acknowledges the request by sending a SYN and an ACK showing its readiness to

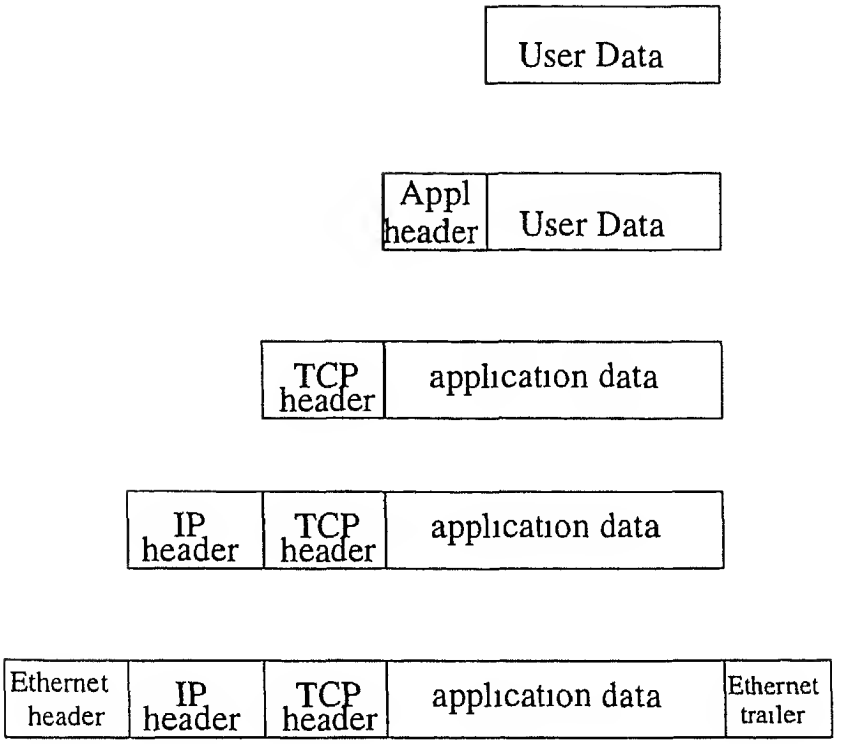


Figure 2 3 Encapsulation of data as it goes down the protocol stack

establish a connection Finally the client acknowledges by sending an acknowledgment If this process is not complete within the time frame determined by the timeouts of both the ends the establishment of connection fails

To handle nasty situations a three way handshake is essential It is obvious that two way messaging is compulsory The third message ensures that the client's timer for reception of the acknowledgment for its request has not expired and it is still interested in connecting In the relatively congenial situation of a LAN such a complicated pro cedure is not necessary because of low delays and high data reliability If the hosts are up the requests and acknowledgments reach in time and the complicated connection establishment procedure of TCP/IP will be a waste of time In reliable networks the connection establishment procedure can be made similar to the initiation of conversation on telephone The caller dials the phone number and the bell rings on the other side indicating the request to make a connection The person who is being called picks up the phone and says 'hello" which is similar to sending an acknowledgment Now the caller can straightaway start the conversation if there is no chance of a wrong number Thus in a LAN, where the probability of delay is negligible it is quite evident that a two way

handshake is sufficient to provide enough reliability in connection establishment Even if something goes wrong during the connection establishment timers may be used to take care of the situation as explained in § 2 4 6

2 4 2 Termination of Connection

Whereas connection establishment uses a three way handshake connection termination uses a four way handshake According to the algorithm the one who initiates the request, sends a FIN signal showing its will to close the connection The other end acknowledges the reception of FIN After the initiator sends the FIN it can not send any more data but the other end can continue to transfer data over the link Finally the other end also sends the FIN to close the connection After the other end acknowledges the request the connection is closed

A four way handshake is essential to close a TCP connection because it is a full duplex connection A TCP connection (which is full duplex) can be considered as a combination of two back to back half duplex connection and its termination is equivalent to terminating two half duplex connections This is similar to termination of a telephone conversation Neither of the two parties hang up the phone before both have finished Before disconnecting the caller and the called will ensure that they have nothing to say by asking if the other has something else to say

In situations where the transaction is unilateral, a two way termination is also sufficient In a unilateral connection the sender will open a connection only if it has some data to transmit When it finishes it will close the connection To ensure a proper termination, the receiver may acknowledge the termination A unilateral transaction is similar to writing a letter In that case the letter terminates without the consent of the reader and is completely at the wish of the writer

2 4 3 Transmission and Reception of TCP Segments

The transport layer makes transport layer frames and hands it to the IP layer The IP layer appends its own header and the hardware header and hands it over to the device layer or the device driver of the operating system The resulting packet that is finally transmitted on the physical medium is shown in Figure 2 3) The frame given to the

IP layer has the transport layer header (that contains the checksum for the segment and the TCP header) and the application layer data The IP layer checks the *routing table* and finds the next hop device for the packet It then fragments the data to satisfy the size requirements of the network of the device (i e size of all packets transmitted on the network should be less than the MTU of the network) and fills the IP header using the information provided by the transport layer and the information generated by itself (i e from routing table) After filling the IP header it appends the hardware header which it obtains from the routing table entry If the required hardware header is not available in the routing table IP uses ARP to obtain the information

The complete packet is handed over to the device driver (the device trailer is put by the device itself and not the IP layer) The packet as composed by the IP layer is now the responsibility of the device layer which finally transmits it on the physical medium

An identical model of layers exists on the receiver side

2 4 4 Acknowledgments

To ensure delivery of segments TCP client (receiver) acknowledges reception To improve the throughput, various techniques are used which do deteriorate the throughput performance in a lightly loaded LAN especially when the data flow is voluminous and unidirectional

To use bandwidth efficiently, TCP *piggybacks* acknowledgments on a data which may be transmitted in case of bidirectional data flow Very often data might not be available for piggybacking the acknowledgment routine waits till it gets data or a timer specifically set for it expires This is known as *delayed acknowledgment* This is advantageous when the data transaction is interactive like for example in telnet, ftp etc However when the data flow is effectively unidirectional an acknowledgment is sent only when the delayed acknowledgment timer expires adding unnecessary delays

Nagle s rule is another approach to increase efficiency Under this rule, small frames are not sent and the software waits for more data to be sent by the application layer After enough bytes are accumulated or a timer set for the purpose expires, a packet is sent

To avoid unnecessary load created by the acknowledgments, an intermittent acknowl

edging may be done i e all packets received need not be explicitly acknowledged Ac knowledgment in TCP is such that whenever a packet is acknowledged all the packets that precede it are implicitly acknowledged Thus explicit acknowledgments for succes sively received packets may be eliminated by acknowledging the last packet in the set This is also called cumulative acknowledgment

2 4 5 Flow Control

TCP is a protocol designed to work on a network as large as the global Internet and hence has strong flow control capabilities It controls flow of data by dynamically changing the window size and the maximum segment size

TCP uses sliding window flow control with dynamic window size The algorithm prohibits the transport layer to transmit a segment beyond a window whose starting point is defined by the last segment for which the acknowledgment has been received The size of window depends on how quickly a packet is acknowledged and this is tracked using an estimate of the round trip time In case there is congestion the estimate of the round trip becomes large and the window size is decreased so that less data is transmitted into the network and this is expected to decrease congestion This round trip time estimate is also useful in setting the retransmit timers because if the expiry time of the retransmit timer is less than the time taken by the acknowledgment to reach the sender there will be continuous retransmission

The estimates of the round trip time have to be done continuously and are effectively overheads in LANs because here the acknowledgment is received fairly quickly and it is rare that the transport layer has transmitted one window of segments and is waiting for the window to slide However, TCP will always make these estimates

Similarly the segment size can also be used for flow control Larger segments increase the load on the network and small segments will increase the number of acknowledgments Many TCP implementations do not send acknowledgments for every segment they receive, and use cumulative acknowledgments instead Although this decreases congestion and improves the end to end throughput, it increases the processing overhead at the receiver For example, Linux uses per octet acknowledgments which improves the throughput but increases the processing overhead

2 4 6 Timers

To cope up situations where in middle of a session one of the hosts stops responding TCP has some timers Apart from these timers there are timers to implement the techniques used to enhance the throughput of the link In the following we discuss the various timers used by TCP

TCP transmitter maintains a timer at the transmitter which is activated when a packet is sent If the acknowledgment is not received before this timer expires it retrans mits a random packet within the transmit window The expiry time of the *retransmission timer* is not fixed and is varied according to the estimate of the round trip time and a pessimistic value is chosen in the beginning

There is a *partial queue timer* which is set when a request to send a partially filled seg ment is submitted The packet instead of being transmitted instantaneously is queued If before the expiry of the this timer enough data is available to make a full segment it will be transmitted Otherwise the partial queue is transmitted

A *delayed acknowledgment timer* is maintained to see if an acknowledgment can be piggybacked on a data packet For this, if data is not immediately available for trans mission, TCP will wait for some time Thus the *delayed acknowledgment timer* is set whenever an acknowledgment is to be transmitted If data is available before this timer expires the acknowledgment is piggybacked on it otherwise the acknowledgment is sent on the expiry of the timer

There is a *keep alive timer* to see that the connection is not idle for a long time If one sender is not sending any data for a long time the receiver uses this timer to close the connection This is useful when a client sleeps while it was being served by a remote server

In real implementations some more timers are used than mentioned above For ex ample Linux uses two more timers One at the IP layer to ensure continuous reception of IP fragments and one general purpose timer

These timers are served by a separate process belonging to the scheduler software This section of the code maintains a list of all the active timers and checks the expiry of all the timers at each clock tick When a timer expires, it calls a function told to it when the timer was initialised and the timer is deleted from the list Longer the list

more will be the load on the scheduler and hence the processor This will in turn effect the performance of other processes In a massive unidirectional transfer over a reliable LAN the partial queue timer and the delayed acknowledgment timer are useless The keep alive timer is also of no use Thus in such cases two timers one transmit timer at the transmitter side and a receive timer at the receiver side is sufficient This will be discussed in more detail in § 4 4 5 and § 4 5 3

2 4 7 TCP State Diagram

The above discussion shows the complexities of the TCP/IP protocol which is inevitable because the protocol is designed to work reliably on a variety of large and small networks The extent of the complexity is evident from the TCP state diagram shown in Figure 2 4 It is a state digram with eleven states and twenty transitions! It is interesting to observe that during a session over a congenial network the set of states of visited will be a small subset of that shown in Figure 2 4 If a state machine were to be designed for such networks, it will be considerably simpler than the one of Figure 2 4 Some states can be altogether removed Some state transitions in the remaining states can also be assumed to be nonoccurant and if they occur a strategy to close the connection may be followed Thus TCP itself can be reduced to a lightweight transport protocol which will work very reliably on small LANs

As will be discussed in chapter 4 a reduced set of states and transitions will be enough to ensure reliability of data delivery on congenial situations like that in LANs This reduction will improve the performance of the transaction and will ensure an efficient use of the available bandwidth

2 5 An Analysis of TCP/IP Implementation Overhead

Many of the overheads in TCP/IP are caused by the particular way in which it is imple mented The implementation is bound to be more complex than the protocol This is because, to implement details of the protocol, some overheads may be added that reduce the efficiency further

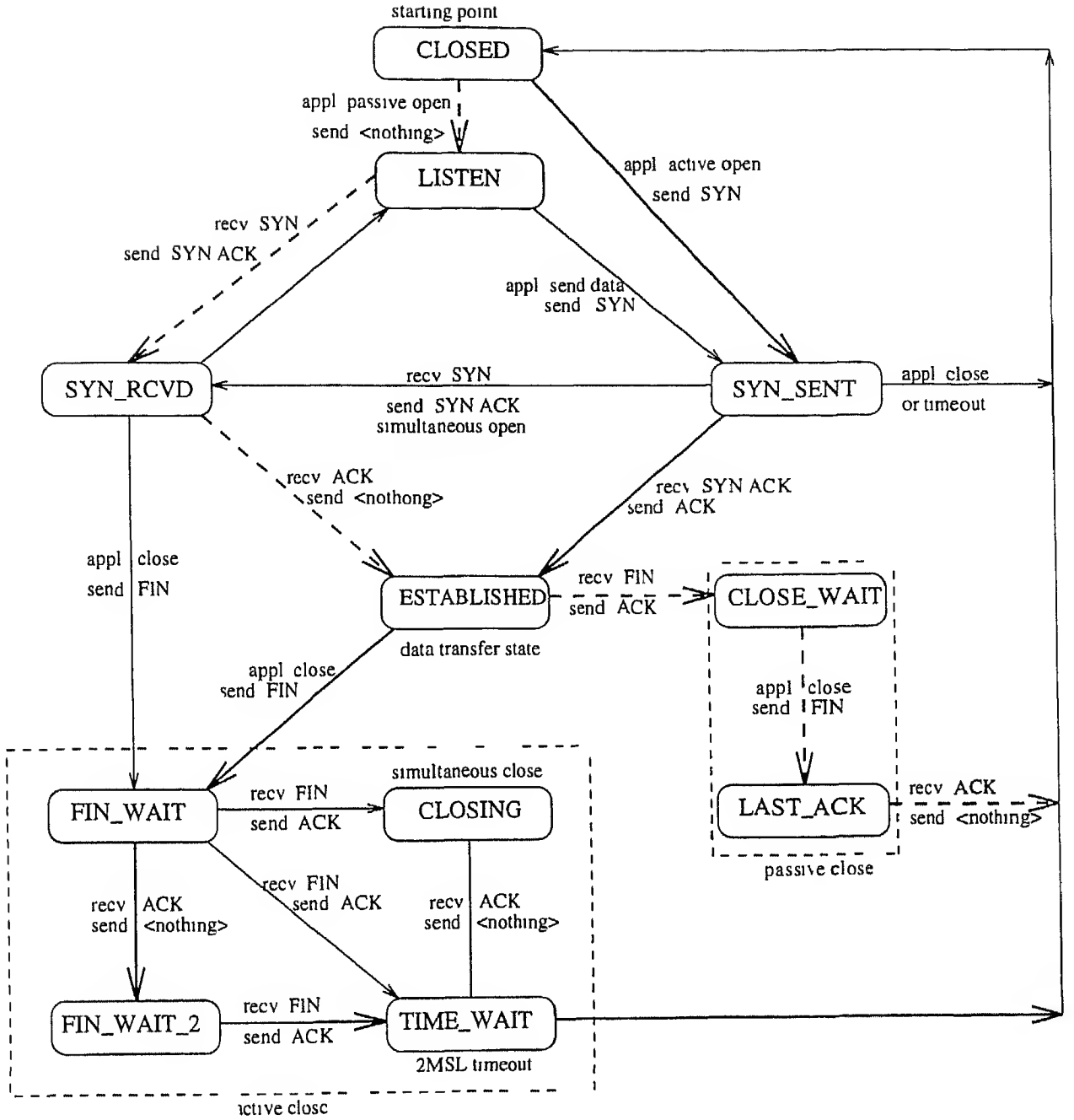


Figure 2 4 TCP state diagram

A study of the Berkelev implementation of TCP as included in UNIX is available in [1] In this paper Clark *et al* count the number of instructions for the normal flow path in the TCP state machine Berkeley implementation uses a buffering scheme in which data is stored in a series of chained buffers called `mbuffs` This buffering scheme is obviously the characteristics of the rather than TCP as a protocol

2 5 1 Input Processing

There are three stages of the TCP processing In the first a search is made to find the local state information (called the Transmission Control Block or TCB) for this TCP connection In the second the TCP checksum is verified This requires computing a simple function of all the bytes in the packet In the third stage the packet header is processed

The calculation of the checksum depends on the raw speed of the environment and the detailed coding of the computation The calculation is done once the whole application layer packet is received

Searching of the TCB can be speeded up by maintaining a TCB cache This leads to a very light algorithm on an average A study showed that on a workstation in general use (opening 5 710 connections over 38 days and receiving 353 238 packets) the single entry cache matched the incoming packet 93 2For a mail server, which might be expected to have a much more diverse set of connections, the measured ratio was 89 8and 121 676 incoming packets)

The packet input processing code has rather different paths for the sender and receiver of data The overall numbers are the following

- Sender of data 191 to 213 instructions
- Receiver of data 186 instructions

Both sides contain a common path of 154 instructions Of these 15 are either pro cedure entry and exit or initialisation For the receiver of the data, an additional 15 instructions are spent sequencing the data and calling the buffer manager, and another 17 are spent processing the window field in the packet

The sender of the data, which is receiving control information has more steps to perform In addition to the 154 common instruction it takes 9 to process the acknowl edgment, 20 to process window, 17 to compute the outgoing congestion window (so called "slow start' control), and 44 instructions (but not for each packet) to estimate the round trip time The round trip delay is measured not for every packet but only once per round trip For short delay paths, where one packet can be sent in one round trip this cost could occur for every acknowledgment Since the Berkeley TCP acknowledges at most

every other packet in a bulk data transfer the cost in this case is 22 instructions per packet For longer paths the cost will be spread over more packets so 22 instructions is an upper bound

2 5 2 Output Processing

The output analysis is somewhat less detailed than the input side To send a TCP packet 235 instructions were used This number provides a rough measure of the output cost but it is dangerous to compare it closely with the input processing cost In fact TCP puts most of its complexity in the sending end of the connection This complexity is not a part of part of packet sending but a part of receiving the control information about that data in an incoming acknowledgment packet

2 5 3 Cost of IP

In the normal case IP performs very few functions Upon inputting of a packet it checks the header for correct form extracts the protocol number and calls the TCP processing function The executed path is almost always the same Upon outputting the operation is even more simple

The instruction counts for IP were as follows

- Packet receipt 57 instructions
- Packet sending 61 instructions

It is not indicated in [1] whether the above are number of instructions in the program or the number of instructions actually executed during the running of the respective programs We believe it is the former because the end-to end delays experienced during a data transaction correspond to the execution time of a much larger number of instructions than those indicated above

The implementation of TCP/IP on Linux is very much similar to the implementation of TCP/IP in BSD It also has data arranged in chain of buffers called `sk_buffs` Most of the implementation overheads that exist for BSD are also existent in case of Linux

Chapter 3

Implementation of Network Module in Linux

3 1 Introduction

Like all other Unix like systems, Linux also provides network access to the application programs through a group of system calls called *socket system calls* These system calls like the file access system calls, provide basic facilities like creating an interface, making and closing connections sending and receiving streams of bytes and various control functions Each network protocol family separately implements these system calls and advertises its implementation using a process of *registration* with the socket interface of the kernel This process of registration makes the socket interface aware of the existence of the protocol family in the kernel and passes on the requests from the application programs to the protocol family Simultaneously the protocol has to make the device layer of Linux also know about its presence in the kernel so that the bottom handler of the protocol, can transfer the packets meant for it that arrive over the device The bottom handler is that part of the kernel which processes the incoming packets from the device and distributes them to the various protocols that are registered with it The protocol family to which an incoming packet belongs is obtained from the hardware header in the incoming packet For an Ethernet packet, this information is contained in the 16 bit protocol field of the Ethernet header The process of registration and referencing of function calls is in the 'top half" of the Linux network module and will be discussed in

§ 3 2 The bottom handler and the registration of the protocols with the bottom half of the network module will be discussed in § 3 3

3 2 The Top Half

The network module of the Linux operating system resides in the `net` directory of the Linux source code The socket interface is provided by `socket c` which serves the requests from the application layer and passes them on to the protocol specific routines as will be explained in § 3 2 3 The structure of network module code is shown in Figure 3 1 The protocol after acting upon the data given by the application layer passes it to the device handlers resident in the directory `net/core` (more specifically in `dev c`) The routine in `dev c` finally calls the device driver to transmit the packet

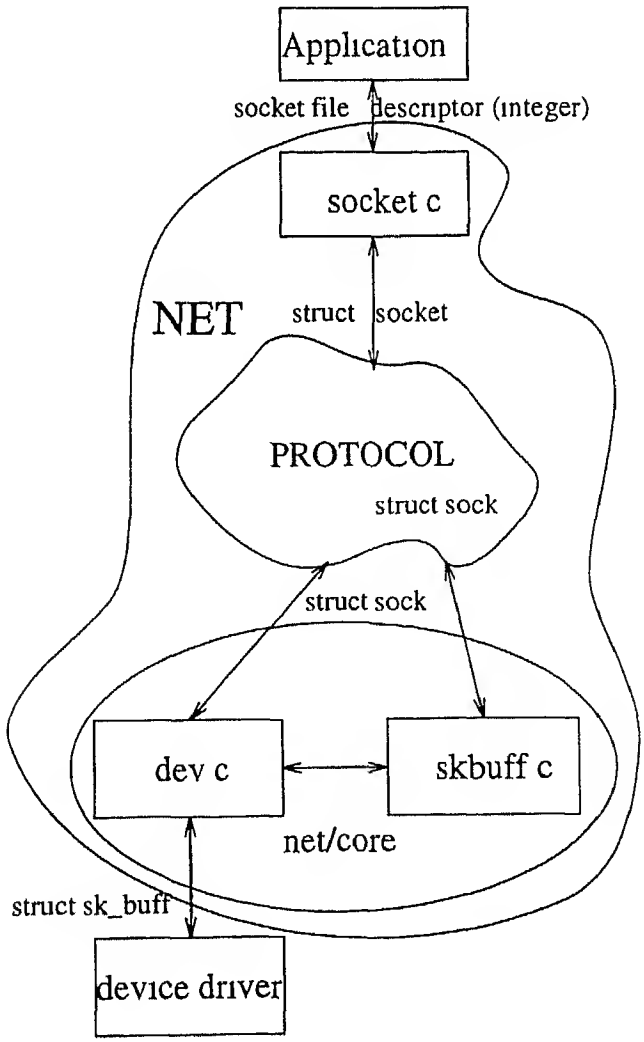


Figure 3 1 Structure of the directory net

On the receiver side the same path is followed The device driver calls the routines of dev c and delivers the data to it The bottom handler residing there distributes the data to the respective protocols The protocols after processing the input data stream hand over the data to socket c and the data is finally passed to the application layer

3 2 1 Registration Procedure

The protocols register themselves through a structure proto_ops which contains an integer identifier used by socket c to identify that particular protocol and a set of pointers to functions which are implementations of the system calls The registration is done by a function sock_register defined in socket c which takes a variable of type proto ops and adds it to a global array maintained by socket c Whenever a socket system call is made for a protocol family, socket c first finds the entry corresponding to that protocol and directs the control to the corresponding function using the pointer fields of the entry (which is of type proto_ops)

There is an initialising routine for each protocol which registers it by calling sock register Initialising routines corresponding to each protocol is available in an array maintained by protocol c These routines are called by sock_init which initialises the entire socket interface sock_init is called when the kernel boots up by the function start_kernel (in init/main c)

3 2 2 Interface to the Application Layer

The kernel provides access to the network through a variable of type struct socket It is known to the application layer through a unique integer for each instance of a *socket* called the *socket file descriptor* (See Figure 3 1)

Whenever a request to create a socket of a certain network address family is made using the system call socket the control goes to socket c socket c then checks that particular family in its database If it finds one, it creates a variable of type struct socket, assigns a file descriptor for it and records this assignment for future use Type struct socket has a field ops of type proto_ops to which socket c assigns the variable of type proto_ops known to it at boot up time Thus, after this assignment socket->ops points to the structure containing all the implementations of the system calls relevant

to the protocol This reference is used by the socket interface when the system calls are made

Finally after all these assignments are made the `create` function of the protocol is called using `socket->ops` i e `socket->ops->create` is called The control then reaches the protocol routine There, relevant initialisation is done the details of which will be discussed in § 3 5

3 2 3 Calling System Calls

After a socket is created in the `ops` field of the `socket` pointers to all the functions are stored When any system call is made the application layer specifies the `socket` by passing on the socket file descriptor returned by the `socket` system call

When a call to `send` a message is made the control first reaches `socket c` `socket c` then finds the `socket` corresponding to the socket file descriptor passed to it After getting the `socket` `socket c` calls the corresponding function from the pointers available in `socket->ops` e g in this case `socket->ops->sendmsg` will be called with parameters which are standard for the interface between `socket c` and the corresponding protocol implementation Thus the job of `socket c` is to direct requests from the application layer to particular protocols This redirection is invisible to the user and the user program has to specify the protocol only while calling the `socket` function to create an interface or a `socket` After creation of a socket all the functions have a similar structure when viewed from the application layer

3 3 The Bottom Half

The bottom half deals with the incoming packets from the device driver The arriving packets are delivered to the respective protocols The transaction between the device driver and the bottom half of the network module is made through a structure `struct sk_buff` This `sk_buff` is given to the protocol to which the packet belongs The identification of the destination protocol for an arriving packet is from the information provided by the protocols at the time of boot up This is done by the initialising function described in § 3 2 1

3 3 1 Registration with the Bottom Handler

Each protocol, at the time of its initialisation gets registered itself with the bottom half of the network module This is done by calling the function dev add pack This function takes a structure of type packet type as a parameter This parameter has a field for a 16 bit identifier for itself (in Ethernet, the identifier is the same as the protocol field of the Ethernet header) and a field for a pointer to a function which has to be called when a packet of the type specified by this type field arrives from the network This function will process the packet for the protocol When the function dev_add_pack is called it adds the information in the above parameter into a list maintained by it for future use Whenever a packet arrives this information is used to call the handler of the protocol to which the packet belongs

3 3 2 Delivery of Packet to the Protocol Handler

All the incoming packets are handled by the function net bh which is interrupt driven The transaction between the device driver and the upper layer is through a data structure struct sk_buff as shown in Figure 3 1 When a packet arrives at the device driver it allocates an sk buff, fills the necessary fields (e g size protocol etc) and calls a function netif rx defined in dev c netif_rx initialises net_bh to be the bottom half handler and marks the interrupt bits Thus net bh is called for every incoming packet

As explained in § 3 3 1 net_bh has access to a list which contains information about the protocol handlers and their type When a packet arrives, it compares the protocol field of the sk_buff with its database If it can find the protocol in its database it calls the corresponding handler and passes the incoming sk_buff to it The sk_buff is now the responsibility of the protocol

At this level, i e , the bottom half of the network module, there can be a possible tap to extract the the incoming packets If in the list of protocols registered with the bottom handler, there is a protocol of type 13 (defined in the kernel code as ETH_P_ALL) the handler of that protocol will be called every time a packet arrives irrespective of the type field in the hardware header This facility is included only to provide a tapping point for all the incoming packets There can be more than one registration with type ETH_P_ALL

3 3 3 Delivery of a Packet to the Device Driver

The previous section dealt with the mechanism by which a packet that has arrived at the device is given to the respective protocol The current section deals with the process by which a packet is given by the upper layer to the device driver

As explained in § 3 2, the device handling routines reside in the directory `net/core` The sending of packets is done by the function `dev_queue xmit` This function performs some checks and tries to transmit the buffer by calling the device driver explicitly if there is no other packet waiting to be transmitted If there is another packet to be transmitted this packet is queued If this packet is being retransmitted it is placed at the head of the transmit queue If the device driver fails to send the packet it is again put in the queue

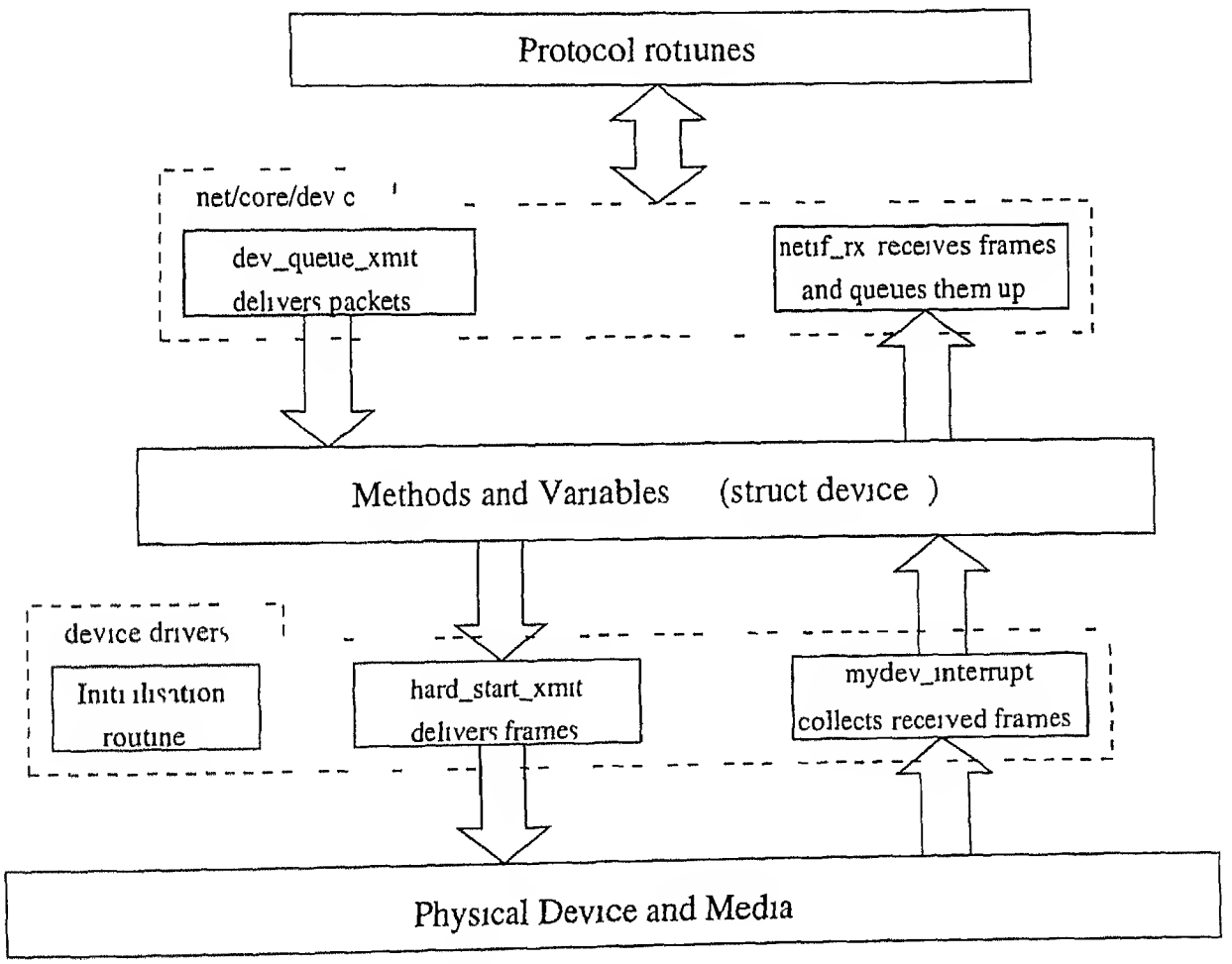


Figure 3 2 The basic structure of device interface

The entire scheme of the device interface is shown in Figure 3 2

3 4 Network Buffer Management

All the buffers used by the networking layers are sk_buff The control for these is provided by core low level library routines available to all the the networking routines sk_buff provides general buffering and flow control facilities needed by the network protocols [2]

The network devices are accessed through a variable of type struct device This device structure contains all the generic information and methods for the network device

3 4 1 The sk_buff Structure

The primary goal of the sk_buff routines is to provide a consistent and efficient buffer handling mechanism to the network module To be consistent, the higher level sk buff and socket handling facilities need to be followed by all the protocols in the network module The sk buff can be queued forming a doubly linked list for sequential processing by the device driver and the protocol An sk_buff is a structure with a block of memory attached with it

There are primarily two types of routines defined in the sk_buff library The first category deals with the manipulation of the list of sk_buff and the second deals with the manipulation of the memory block associated with it

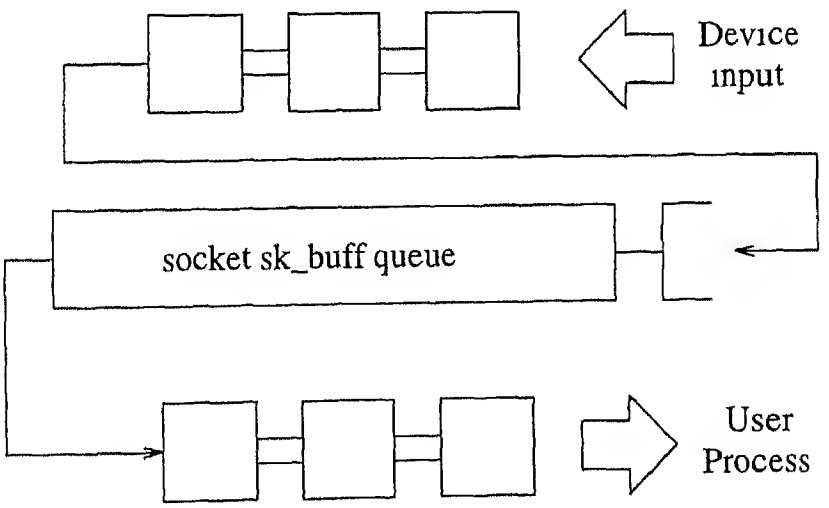


Figure 3 3 Flow of sk buff while sending

The first type of routines, those for manipulation of sk_buff lists, are used in both the receiving and sending of data Whenever data is to be sent, the protocol layer allocates an

sk buff, fills it and queues it into a list of sk buff maintained by the device structure Each device has three queues which hold packets of different priorities (low delay high throughput and high reliability) The sk buffs are taken from the head of the device queue and transmitted by calling the device driver This is shown in Figure 3 3 On the receiving side, every incoming packet is placed into the receive queue of the sock for the corresponding connection to which the packet (or the sk_buff) is destined This is done by the protocol handler as explained in § 3 5 This sk buff is taken out from the receive queue of the sock when a receive() is called from the application as shown in Figure 3 4

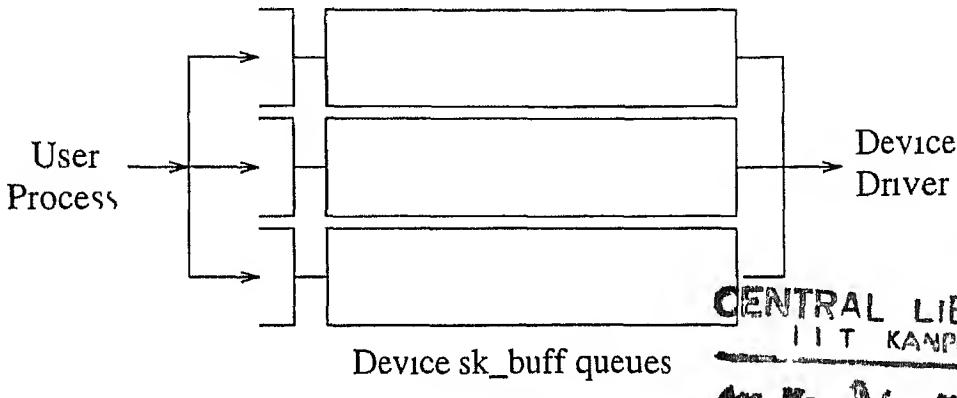


Figure 3 4 Flow of sk_buff while receiving

The second type of routines are those that deal with the manipulation of the memory block attached to the sk buff Every sk_buff has three parts viz the head room the data area and the tail room These are defined by pointers held in the sk buff structure On allocation of an sk buff of a certain size, the entire memory block is occupied by the tail As the various headers and the data are filled the data area grows and the others contract This is primarily done by the routines provided in the buffer library

3 4 2 The device Structure

All Linux network devices follow the same interface although many of the functions available in that interface will not be needed for all the devices An object oriented design is used and each device is an object with a series of methods that are filled into a structure (the device structure) Each method is called with the device itself as the

first argument

The generic information and the methods for each network device are kept in the device structure To create a device most of the fields for the methods needs to be initialised Each device is identified by a string pointer *name* like `eth0 eth1 tr0` After filling the methods and variables, the device initialisation routine advertises the device by calling the function `register netdev()`

The `device` structure contains a block of parameters used to maintain the location of a device within the device address space of the architecture e g fields to hold `irq` number, `dma` channel `base_address`, `mem start`, `mem_end` etc A group of parameters are provided to be used by **net-tools** which is a package consisting of tools like **ifconfig** which are used to set and get various device parameters like MTU protocol address various flags capabilities etc There are a few parameters that are used by the protocol layer e g `hard_header_len`, `pa_addr` (the protocol address) etc

The methods attached with the device consists of the basic routines to set up the device, fill hardware header (`dev->hard_header()`) transmit a frame (`dev->hard start_xmit`) etc These functions are used by the lowest level of the protocol layer to send and receive frames Together these methods comprise the *device driver*

3 5 The Protocol Layer

The protocol layer lies below `socket c` and above the low level device access routines defined in `dev c` as shown in Figure 3 1 A protocol consists of a set of methods defined in accordance with the rules laid out by that protocol These methods are contained in the `proto_ops` structure filled by the protocol initialising routine as explained in § 3 2 1 and are called when a system call is made as explained in § 3 2 3

3 5 1 The sock Structure

The system call, as defined in `socket c` calls these methods with `socket` as a parameter Within the body of the protocol, the data structure used is the `sock` structure as shown on Figure 3 1 When the socket is created (or opened) a `sock` is allocated and the pointer to it is stored in the `data` field of the `socket` The `sock` structure contains information

relevant to the protocol e g window size socket MTU timers sequence number of the packet sent sequence number of the last packet for which an acknowledgment has been received, etc

When the socket is opened the system call eventually calls the `create()` of the protocol to which the socket belongs The `create()` routine assigns a `sock` for the `socket` as explained earlier and initialises the various fields The sequence numbers are initialised to ensure proper start e g TCP initialises the sequence number of the last packet sent as a random number following RFC 793 and the sequence number of the last packet for which acknowledgment has been received as one less The window size is fixed as two (slow start) The MTU is set to 576 All the timers are initialised

Apart from these data fields the `sock` has a `prot` filed which has methods to implement various transport layer protocols on the same network layer protocol When the `sock` is initialised, the relevant variable is filled in the `prot` field of the `sock` For example, the TCP callbacks are filled in `tcp prot` and when a TCP socket is opened `socket->sock->prot` is assigned to `tcp prot` Now, when a send is called on a TCP socket, `socket->sock->prot->sendmsg()` is called

The `prot` structure has an array of `sock` `sock array` which contains all the active opened connections Thus `tcp prot->sock array` has a `sock` to which an incoming packet has to be delivered Entry corresponding to a socket into this array is made while `bind()` is called on that particular socket

The `sock` structure contains three lists of `sk_buff` The `write` queue the `receive` queue and the `backlog` queue The `write` queue consists of those `sk buff` which are not being sent because of the limitation imposed by the window size Whenever an acknowledgment comes, an `sk_buff` is sent from this queue to the device The `receive` queue holds the packets that have arrived from the device These `sk_buff` are added by the protocol handler `sk buff` are taken out from this queue and the data is copied in the application area The `backlog` queue holds those packets which have arrived but cannot be put in the `receive` queue

3 5 2 Network Layer Implementation

The network layer provides primitive support for sending and receiving datagrams This support is unreliable and is used by the transport layer which takes care of the reliability issues This section makes an overview of the implementation of network layer giving special references to implementation of the IP layer (Note that the network module of Linux is designed to suite the requirements of TCP/IP)

The transport layer after filling its header calls the routine to fill the network layer header After these headers are filled control passes on to the network layer to put the packet on the device The network layer does a translation from protocol address to hardware address To do so it uses its own methods e g IP maintains a routing table entry which it looks up while a packet has to be sent To speed up this process a field is there in the sock structure to hold a routing table entry which is used as cache Before looking the routing table it first compares the destination address with the routing information stored in this cache If it is relevant it is used otherwise the routing table is checked and the entry read recently is stored in the cache Using this information, the network layer fills the hardware header It allocates an sk_buff, fills the hardware header the network layer header and the data (which contains the transport layer header) into the sk buff Finally it sends the packet to the bottom half

If the data block provided by the transport layer is larger than the device MTU (the device to which the packet has to be sent is found from the routing table) it has to be fragmented into small pieces of size less than or equal to the device MTU Thus the network layer breaks the data block and send them successively on the device On the receiving side, before delivering the packet to the transport layer these fragments are reassembled in order

After reassembly of the fragments, the network layer calls the transport layer protocol handler based on some information provided by the network layer header e g the IP header has a field for the protocol(TCP, UDP ICMP etc) The frame now becomes the responsibility of the transport layer

3 5 3 Transport Layer Implementation

The transport layer is built completely over the network layer It adds extra reliability and controls the flow of data The transport layer divides the application layer data into segments and fills up its header in these segments If the size of the window allow the transmission of this segment it is delivered to the network layer otherwise it is added to the `write` queue as explained in § 3 5 1 These segments are tagged by a sequence number which is used to ensure a delivery of the segment If reception of a segment is not acknowledged for a certain amount of time retransmission is done to measure this time there are timers associated with each **socket** on whose expiry retransmission For retransmission each transport layer protocol has its own algorithm (TCP uses Go back n strategy)

On the reception side the network layer gives the assembled segment to the transport layer The transport layer finds the segment sequence and correspondingly sends or withholds the acknowledgment Using the sequence number information it arranges the segments in order and throws away multiple copies of the same segment

Once an ack is received, segments are released from the write queue as explained in § 3 5 1 according to the algorithm followed by the protocol

The transport layer controls the amount of the flow of data by controlling various parameters associated with the connection e g window size etc

These implementations complete the whole map of **a protocol** from the device layer to the transport layer The object oriented approach of Linux makes all the protocols look same from the application layer The internal implementation of the protocols also follow the object oriented approach and thus lead to some kind of similarity in the basic structure Moreover existence of library of generic routines to handle common aspects like buffers, sockets timers, queues etc provide a modular and layered structure to the implementation

Chapter 4

The Lightweight Communication Protocol

From our discussions in the preceding chapters two things are evident Firstly the general purpose TCP/IP protocol will cause a considerable overhead when used for distributed computing applications especially in LAN environments Secondly Linux provides enough flexibility and a clean modular mechanism to implement another communication protocol that can be used by any application The normal implementation of Linux has TCP/IP and Unix protocols in its kernel In this chapter we discuss the design and implementation of a lightweight communication protocol specifically for use by distributed applications in Linux based system on LANs We call this protocol Light Communicator LightCommunicator is not intended as a general replacement for the TCP/IP stack rather as a replacement in the specific environment of distributed computations over LANs

In the next section we discuss the distributed computing environment In § 4 2 we discuss the desirable characteristics of a communication protocol for the target distributed computation environment In § 4 3 we present the design of the LightCommunicator stack The layers in this stack are explained in §4 4 and §4 5

4 1 The Distributed Computation Environment

In the following we characterise the distributed computation environment and identify the features that make substantial simplifications to the communication protocol stack possible We assume that the distributed systems are interconnected through an off the shelf multiple access LAN like for example a 100Mbps Ethernet We first describe the characteristics of this LAN in terms of its topology throughput and error performance We then characterise the nature of data transaction in the certain kinds of distributed computations for which this protocol has been designed

4 1 1 The Underlying Network

In the design of LightCommunicator we make the basic assumption that all the nodes are on the same network, i e , the messages do not cross routers This is a reasonable assumption because, typically, routers introduce considerable delays in the communication path and will deteriorate the performance of distributed computations

In a LAN environment typically, we can expect a *friendly* behaviour of the network towards the packet that are being floated This is because of many reasons Firstly chances of the network introducing errors is very low and can be neglected and the checksum of the MAC layer packet trailer is sufficient This saves us from introducing an extra error detection/correction overhead This is because the Ethernet error detection mechanism is known to be quite reliable Secondly the packets always received in order This is because there is only one path between the sender and the receiver This feature helps us to detect a packet loss If a packet arrives before its predecessor arrives the preceding packet is lost

The round trip delay from the sender to the receiver is very low and we can expect a prompt response from the peer protocol at the destination This allows us to avoid unnecessary delays by waiting for long when acknowledgments are not received at the sender and having low timeout values for the retransmission timer This is reasonable because the end to end delay is only the time taken to transmit on the network and the device driver processing at the sender and the receiver For example if a packet is not followed by another one within a small interval, we can assume that there is something

wrong and corresponding corrective action can be initiated

4.1.2 The Transactions

In a distributed computation environment, typically there is intensive data transaction and each transaction is predominantly unidirectional. This means that the overhead introduced to increase bidirectional throughput by piggybacking acknowledgments on data or waiting for a segment to be filled when a partial frame is ready at the sender can be avoided. Also most distributed computing is done using message passing libraries which usually follow a master slave architecture. In this architecture a typical transaction will comprise of opening of a connection, unidirectional transfer of data and then closing of the connection. Since communication is not interactive like for example in telnet, the data flow is predominantly unidirectional and only one side is sending at a time. Such a connection should ideally be controlled by the sender only.

In the environment that we are considering, we can assume that if a host is not responding promptly, it is down. This is reasonable because a packet cannot be blocked at any place in the network because there are no routers between the sender and the receiver to store the packets.

Any kind of flow control is unnecessary because there is no router between the sender and the receiver. Also, the Ethernet collision resolution mechanism is an effective flow control scheme. This means that a lot of the flow control mechanism overhead that TCP has to live with can be avoided.

4.2 Defining the Characteristics of a LightCommunicator

The lightweight protocol described here is meant to be a replacement for the TCP/IP stack in distributed computation applications. Thus the biggest requirement is that the TCP/IP interface be mimicked by the LightCommunicator stack i.e., LightCommunicator must simulate the interface provided by TCP/IP to application programs both in terms of the functions called and the interpretation of the parameters passed to/from these functions. In this respect a considerable amount of the compatibility is ensured by

the implementation of the socket interface What is required is that the implementation of the methods stated in § 3 2 3 should be identical to that of TCP/IP externally and they should take the same arguments in the same format as the corresponding methods in TCP/IP implementation The following example illustrate this requirement When TCP/IP is used for communication, the application programs address the destination using IP addresses Therefore the protocol must accept IP addresses and port numbers If any other addressing mechanism is used the conversion should be internal to the protocol code Similarly, on the receiving side it should report to the application layer with IP addresses and port numbers instead of its own addressing scheme For the same reason observe that implementation of any facility to the application layer other than that provided by TCP is redundant because that facility is not going to be used

Also, note that there are many utilities which have no meaning when TCP is replaced by some other transport layer protocol and IP is replaced by some other network layer protocol For example `setsockopt()/getsockopt()` functions have options to set and get the window size But if the transport layer protocol does not use window flow control this option has no meaning Similarly routines to control segment size also lose their significance and some of the networking layer routines are also rendered meaningless However these should not be left unimplemented, but should be implemented in a relevant way For example, the function to get the window size should return some sensible value so as to not confuse the application program

4 3 The Stack

LightCommunicator uses stop and wait with selective negative acknowledgments to provide reliability The application data is broken into fragments and the fragments into packets Packets corresponding to a fragment are sent back to back without waiting for acknowledgments After a full fragment has been sent the receiver sends requests to retransmit the packets which have been lost If all packets are received correctly, an empty request is sent On receipt of an empty request, the sender then proceeds to the next fragment Timer are used to tackle unusual situations

The organisation of the LightCommunicator lightweight communication protocol fol

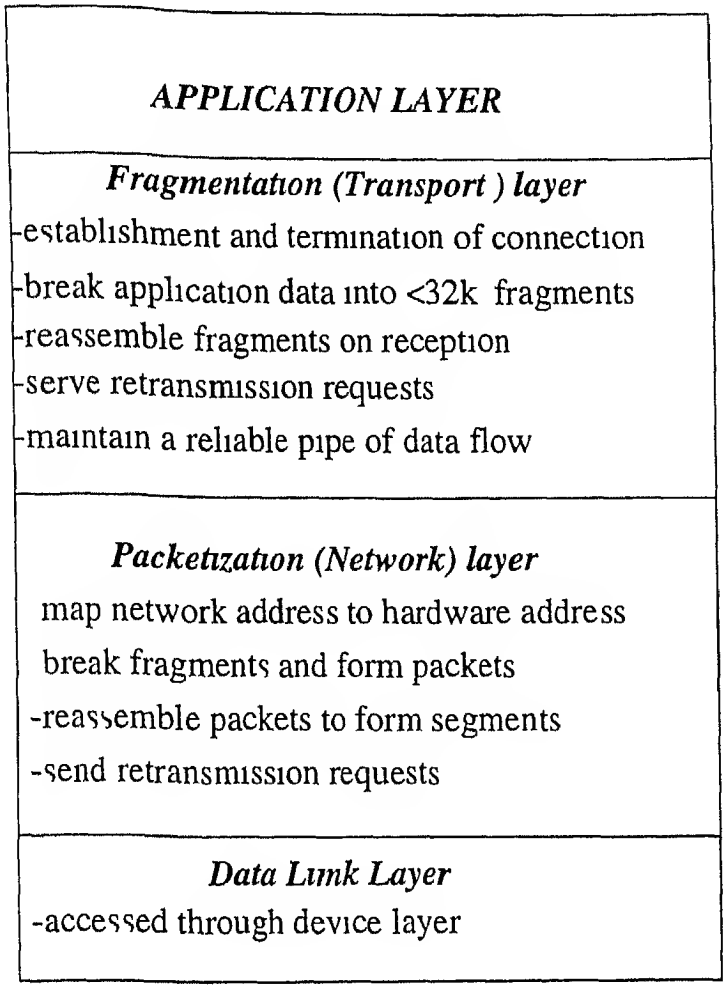


Figure 4 1 The stack used by LightCommunicator

lows that of TCP/IP The network layer is replaced by a Packetisation layer and the transport layer by a Fragmentation layer The functions of these will be described later The device layer used by TCP/IP is also used by LightCommunicator and its implementation in the Linux kernel can be used (The device layer includes the device drivers and the routines to access device drivers) Figure 4 1 shows the complete stack and the operations performed in each layer

The advantage of using the same stack as that of TCP/IP is that the implementation of the network module in Linux supports such a structure Moreover, splitting of application data in two levels helps in maintaining a continuous flow of data and an efficient management of acknowledgments and retransmission of packets and fragments is possible

A close look at the LightCommunicator stack will show that the stack does violate

the peer to peer structure recommended for layered communication architectures For example, the retransmission request or *nacks* are sent by the Packetisation layer of the sender and these are served by the Fragmentation layer at the receiver This is done to improve the throughput as will be explained later

The various modules of the protocol are organised as shown in Figure 4 2 There is only one module constituting the Packetisation layer but the Fragmentation layer is made up of three modules – a fragmentation module for fragmenting the application data into 32 KB fragments and transmit them a connection module to deal with establishing and termination of connections, an acknowledgment management module to serve acknowledgment which is different from the data transmission part of the fragmentation module but a proper synchronisation is maintained between these two modules

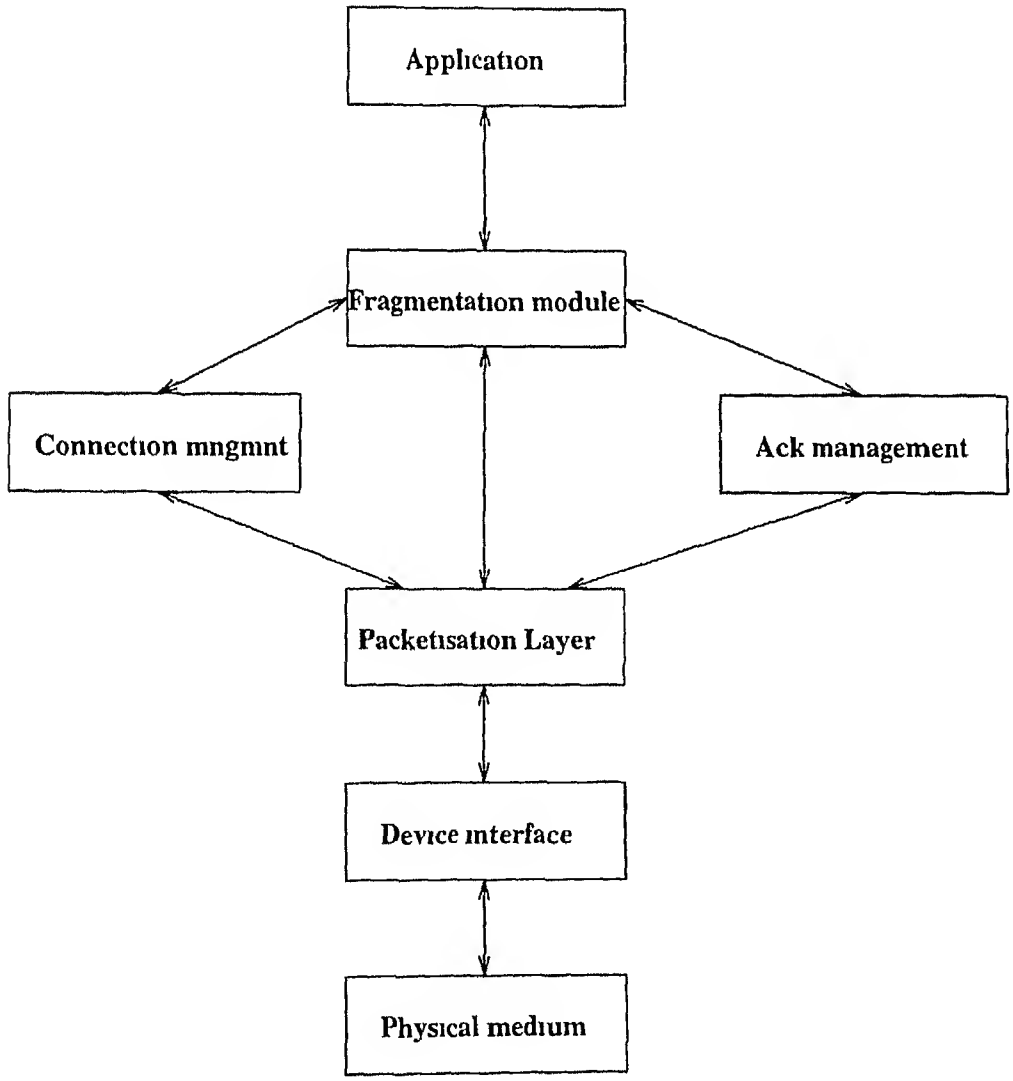


Figure 4 2 Organisation of various modules constituting the protocol

4 4 The Packetisation Layer

As shown in Figure 4 1 the Packetisation layer deals with translation from network addresses to hardware addresses In fact this layer does two levels of translation Since the Fragmentation layer gets the IP addresses from the application it supplies the Packetisation layer with the IP addresses The Packetisation layer translates this IP address to the LightCommunicator address and fills the LightCommunicator header Finally it gets the hardware header from a routing table maintained by it and fills the hardware header The hardware addresses cannot be used by LightCommunicator because this information is not handed down by the device layer of Linux when a packet is received The application will need to know the source of the received data Thus to preserve source information LightCommunicator has its own addressing mechanism

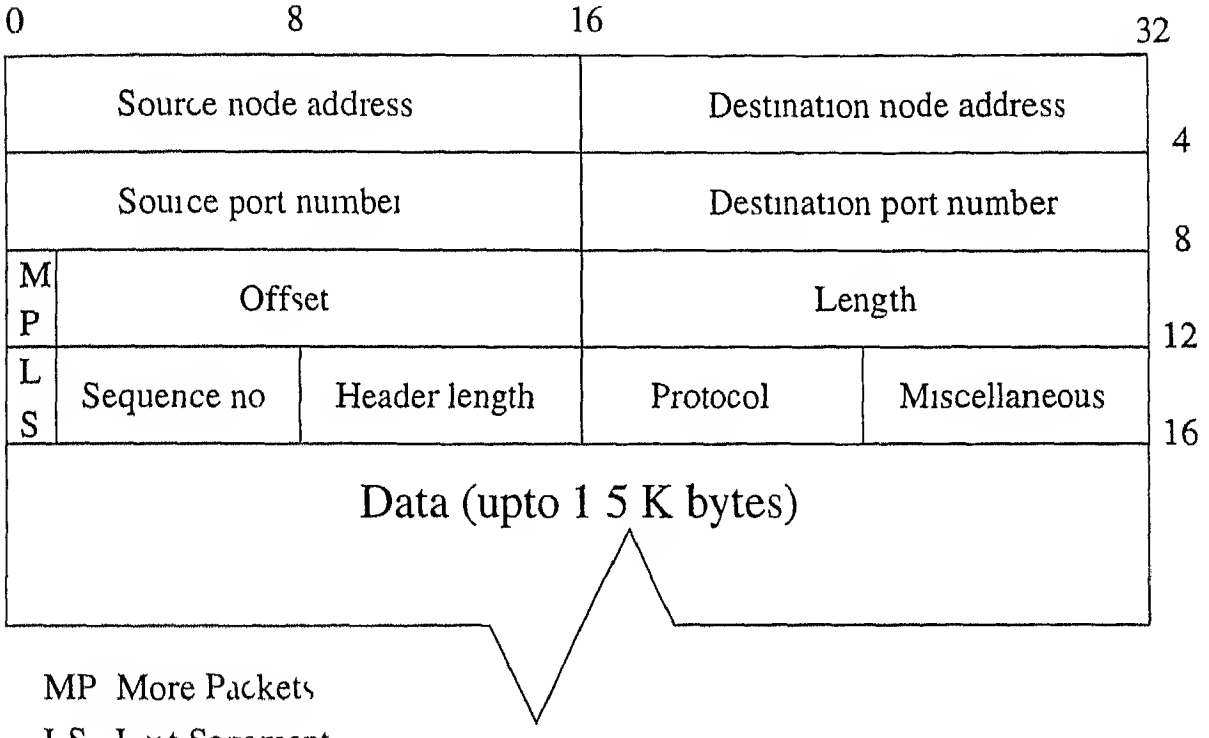


Figure 4 3 The 16 byte LightCommunicator header

4 4 1 The Header

LightCommunicator uses a 16 byte header which incorporates information corresponding to both the network layer and the transport layer of the OSI stack The header structure

is shown in Figure 4.3. The two node addresses are the protocol addresses of the two nodes involved in the communication. The two port numbers are the source and destination port numbers as used by TCP. The offset field gives the position of the particular packet from the starting of the transport layer fragment. There is one bit MP (more packets) indication in the offset field to signify that more packets follow the packet in the current fragment. This field is set in all packets belonging to the transport layer fragment except the last packet. The length field specifies the length of the whole packet (including the header). The sequence number will be discussed in § 4.4.5. The header length gives the length of the header. This is not necessary for present implementation, but it will be needed once the IP options have to be included.

4.4.2 Packetisation of Transport Layer Fragments

The width of the offset field of the protocol header is 15 bytes which imposes a limit on the transport layer fragment size of (32+1.5)K bytes (32 KB due to the size of the offset field in the packet header and an extra 1.5 KB that can be accommodated in the last packet). However this would lead to a 136 byte packet for the last one in a fragment. To avoid this, the maximum size of a fragment has been reduced by 136 bytes to 34132 bytes. This makes a total of maximum 23 packets in a fragment. All these calculations have been done assuming that the device MTU is 1500 bytes and the header length of 16 bytes.

The Fragmentation layer requests the Packetisation layer to send a fragment of data. If the size of the fragment is larger than the device MTU, the Packetisation layer breaks it up into smaller packets. The initial packets are made of size equal to one header length less than device MTU. The residual bytes are adjusted in the last packet.

After the fragment is handed over to the Packetisation layer, it packetises the fragment and sends these packets in the reverse order i.e. it will transmit the last packet first and the first packet corresponding to that fragment last. The advantage of this strategy is that the receiver can know the size of the entire fragment at the arrival of the very first packet belonging to that particular fragment. The packets, when floated on the network are received and handed over to the protocol as explained in § 3.3.2. The Packetisation layer at the receiver composes the segment as will be explained in § 4.4.3 and hands it

to the fragmentation module

4 4 3 The Reassembly of a Fragment

At the receiver there is a queue for each fragment that is being received Note that corresponding to a connection there is only fragment being received at any time These queues are linked into a global list A queue for a fragment is created when the first packet of the fragment arrives Whenever a packet arrives the queue for that fragment (identified by the protocol header) is searched in the list If it is found the incoming `sk_buff` is added to the queue if it is not found a queue is created and the `sk_buff` is added to that

When a packet with offset 0 arrives, indicating all the packets of the fragment have been transmitted, a reassembly of the fragment is attempted If all the packets are available in the queue, the fragment is composed and given to the upper layer If there are packets missing, a retransmission request is sent for all packets which are missing The details are explained in § 4 4 4 If packets are delayed or experience MAC layer errors timeouts occur to maintain the correctness of the protocol The use of these timers are explained in §4 4 5 and §4 5 3

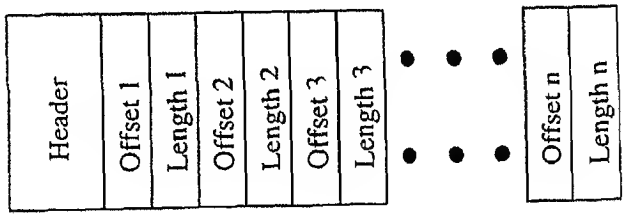


Figure 4 4 The structure of a retransmission request message

4 4 4 The Retransmission Request

If the reassembly fails the receiver sends a retransmission request to the sender The requests are of the form shown in Figure 4 4 For every hole in the reassembled fragment the starting point of the hole and the length of the hole is sent The offset is found from the end of the previous packet which was received and the length is found from the offset of the next packet which is received A retransmission request is identified with a different `protocol` field in the header These requests will be served by the Fragmentation layer of

the sender (§ 4 5 2) and this violates the tenets of layered communication This is done because the information about the loss is first available at the Packetisation layer and there is no point in climbing up the stack to send the requests and unnecessarily delaying the procedure Also, the packets are available only with the Fragmentation layer and there is no point in having the Packetisation Layer asking for them

After the transmitter receives the requests for retransmission it will resend those packets whose retransmission is requested When these retransmitted packets arrive they are added into the queue as a normal packet and assembly is attempted again

The above procedure cannot be done indefinitely There is a counter maintained for each queue which keeps track of the number of retransmission requests sent for each queue Once this number exceeds a predefined threshold, steps are taken to close the connection To do this, the process that has requested the receive has to be aborted i e *the top half* of the network module of Linux For this purpose a packet to abort this process is composed and is put in the `receive` queue of the `sock` for that connection When the top half sees this packet in its queue it will just return without doing anything

4 4 5 The Receive Timer

Situations might arise when the transmitter may fail and the receiver is waiting to receive packets and fragments To not do this indefinitely a receive timer has to be maintained Since most of the input processing is done at the Packetisation layer it is convenient to maintain the timer in this layer This timer has three different timeouts for three different states The first state corresponds to the situation when the receiver is receiving a fragment and has not received it completely Since in this transaction there is no handshake between the transmitter and the receiver and the transmitter just pumps in packets in a stream this state has the lowest timeout value In our implementation we set this timeout to be one second On expiry of this timer, the timer function attempts a reassembly of the fragment being received Since the fragment is incomplete a retransmission request is generated and transmitted as explained in § 4 4 4

Once the whole fragment is reassembled (successfully or otherwise) an indication has to be sent to the transmitter and the next fragment will be obtained when this indication is received at that end Therefore this timeout should be at least double that

of the previous one On expiry of this timer a reassembly is attempted

Once we receive the final fragment corresponding to a message a large timeout is assigned to this timer and will serve as a *keepalive timer* so that a connection which has been inactive for a long time can be closed On expiry of this timer the connection is closed

From the above it is evident that at the receiving side having the timer in the Packetisation layer is better because it is here that the the maximum amount of information about the status of a connection is available

4 5 The Fragmentation Layer

The reliability of data transaction is ensured by the *Fragmentation layer* This layer consists of three modules as shown in Figure 4 2 The skeleton is provided by the *Fragmentation Module* which is responsible for breaking up the user data in fragments of 32 Kbytes and hands them to the Packetisation layer for further processing The *Connection Management Module* establishes and closes a connection The *Acknowledgment Management module* deals with the incoming acknowledgments This division of tasks makes the code clean and modular and reduces some of the processing overheads These modules are explained below

4 5 1 The Fragmentation Module

This module receives application data and the destination IP addresses as its input The primary objective of this module is to ensure reliable delivery of this data to the receiver It uses a **stop and wait with selective nacks** from the receiver at the end of a fragment to ensure reliable delivery of data

The application data is divided into fragments of 32 Kbytes and each fragment is passed to the Packetisation layer where it is further broken into packets and sent over the network The Packetisation layer receives the destination address, the length of the fragment and a pointer to a callback routine to copy the data bytes from the application area to the kernel area After giving the fragment to the Packetisation layer the process blocks and waits for a retransmission request from the receiver If it gets an empty

request, indicating successful receipt of a fragment it proceeds to the process the next fragment If the receiver asks for selective retransmission only those *packets* are sent which are being requested and not the whole fragment At this point also there is violation of the basic layered structure The aim of this violation is to improve the efficiency of the transaction The acknowledgments cannot be served by the Packetisation layer because this would necessitate multiple copies of the application data

4 5 2 Retransmissions

Retransmissions by the transmitter are based on the information provided by the negative acknowledgments sent by the receiver From the retransmission request the transmitter obtains the offset and the length of the data to be sent There can be multiple packets in the fragment that need to be retransmitted

Retransmissions are handled by the same routines as that used in first time transmissions The only difference is in the callback mentioned in § 4 5 1 In the callback if it is a retransmission the `offset` field is changed offset of the hole indicated by the retransmission request This offset is obtained by adding the offset provided by the retransmission request to the offset set by the Packetisation layer The MP bit is also modified accordingly After completion of a retransmission a counter that counts the number of retransmissions is incremented When this counter exceeds a threshold if still negative acknowledgments continue to be received serious network error is assumed and the connection is closed

4 5 3 The Transmit Timer

The transmit timer is placed in the Fragmentation layer because most of the processing in the sending of data is done in this layer Unlike the receive timer it operates only in one mode This timer is used to know the health of the connectivity to the receiver If, after a fragment is transmitted, the acknowledgment is not received before timeout, the whole segment is retransmitted If a timeout occurs on the retransmission, serious network error is assumed and the connection is closed

4 5 4 Acknowledgment Management Module

The acknowledgments are identified by the `protocol` field in by the acknowledgment packets The protocol handler recognises an incoming packet as acknowledgments and queues them into a queue which is different from the one meant for the data packets (obviously!) The routine that serves the acknowledgments will wait for a retransmission request after the fragment is handed to the Packetisation layer It pops the request from the queue and takes further action as explained in § 4 5 2

4 5 5 Connection Management Module

LightCommunicator like TCP, is a connection oriented protocol Before starting the transaction, a connection has to be established This connection has to be closed after the transaction is complete

The connection establishment is a two way handshake This is sufficient because the round trip delays are low There is an interface from this module to the application like in the Fragmentation module

This module is the implementation of the system calls `listen()` `accept()` `connect()` and `close()` The application, when it wants to transmit data sends a connection request to the intended receiver This request is identified from the `protocol` field After sending the request, it waits for an acknowledgment of the request On the receiver side, after the reception of the connection request an acknowledgment is transmitted and connection is declared to be established When this connection acknowledgment is received at the sender, the connection is fully established

Similarly, closing of a connection is also a two way handshake and follows the same procedure as that of opening of a connection This is fairly reasonable because the transaction is predominantly unidirectional This type of one way close does not prohibit a full duplex transaction but only prohibits the user to do a `receive()` after a `close()`

4 5 6 The Other System Calls

Apart from the above modules, there are a number of system calls like `dup()` `bind()` etc that are implemented in the same way as in TCP/IP

Chapter 5

Performance and Conclusions

5.1 Performance

We now present the results of delay measurements on LightCommunicator and TCP/IP

Messages of different sizes were transacted over a lightly loaded network and the end to end delays were measured. The following experimental setup was used. A simple client server program sends a message 20 times using LightCommunicator and TCP/IP. These messages are sent to the server which finds the source address from the LightCommunicator (or TCP/IP) header and echoes the message back to the source. The delay at the source from the instant at which it is submitted to the kernel is measured for each transaction. The mean and variance of these delay measurements are obtained. The experiment is repeated for 20 different sizes of the message. The delay measurements use the `clock()` function of Linux. This function provides the active time taken by the process and its children. For our experiments this does not measure the total CPU load of the protocol but rather gives us the times for which the process is not blocked. It may be noted that TCP/IP generates a higher CPU load than LightCommunicator. For example, TCP adds five timers per full duplex connection putting a relatively larger load on the scheduler whereas our implementation of LightCommunicator uses only two timers. Similarly the bottom handler is also an independent process in itself. There are many other small overheads which are not captured by this experimental setup. Thus the results that we report from our experiments does not capture other advantages of LightCommunicator over TCP/IP. The real test of the utility of LightCommunicator will

be to use it in a distributed computing message passing library like MPI

The measurement results are shown in Figures 5 1 and 5 2 Figure 5 1 shows the sample mean of the delay from the 20 transactions for each message size for both Light Communicator and TCP/IP and Figure 5 2 shows the sample coefficient of variation (ratio of standard deviation to mean) for both the protocols

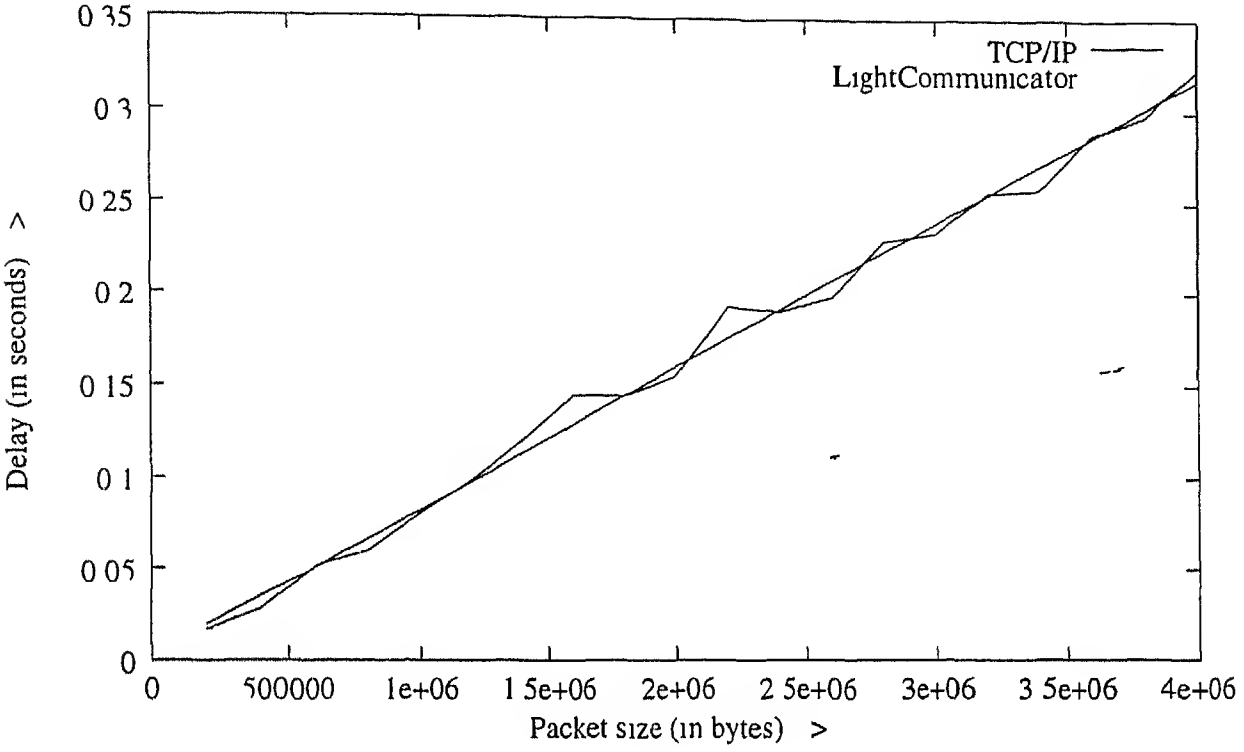


Figure 5 1 The means of the delays for both protocols

The results show that the coefficient of variation of TCP/IP is considerably larger than that of LightCommunicator Considering this with the fact that the mean delay in LightCommunicator is lower the delay variance for TCP/IP is much higher This increased variance of TCP/IP can be attributed to various paths that a message can take through the protocol stack at the source and the destination The difference in the route of navigation along the code may be a reason for a larger variance

If we fit a straight line to the sample mean graphs we get following results

TCP/IP $mean_{TCP}(x) = 0\,088x + 0\,003$ where x is packet size in Kbytes

LightCommunicator $mean_{LC}(x) = 0\,043x + 0\,001$ where x is packet size in Kbytes

A linear fit is sufficient because an attempt to fit a quadratic curve results in a very low coefficient of the second degree term The slope of the line of least square fit of

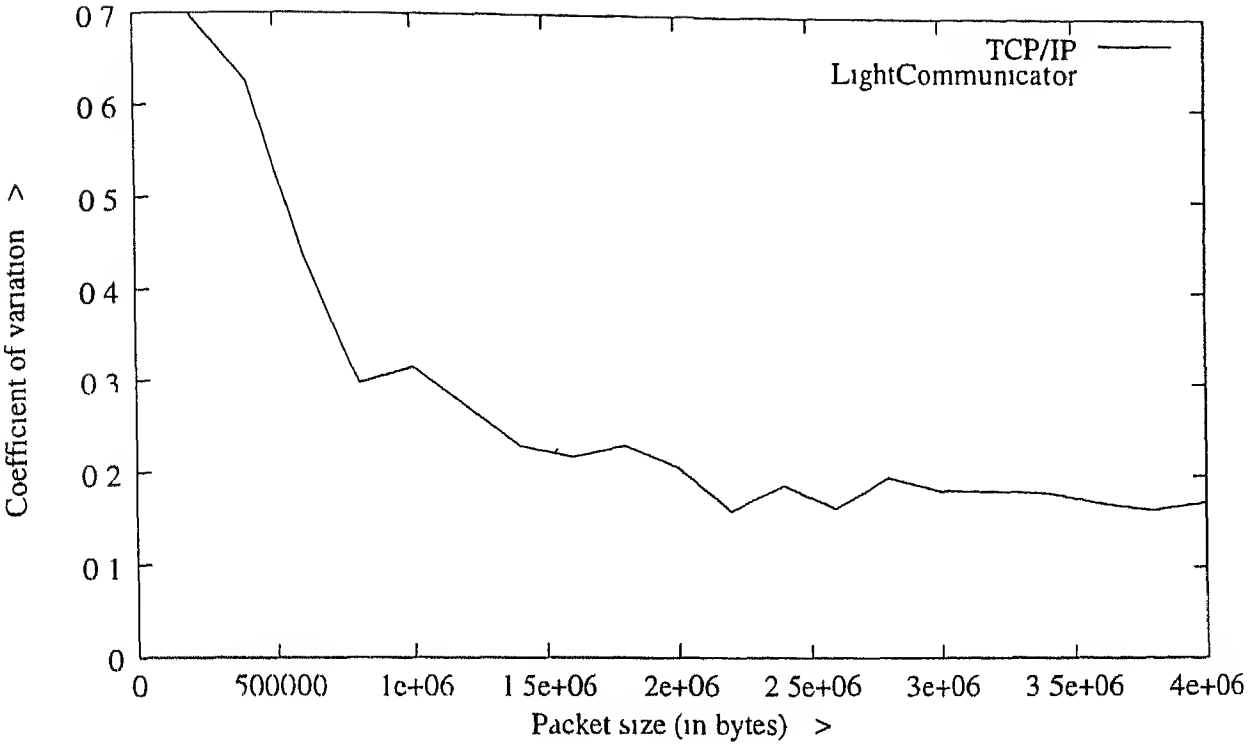


Figure 5 2 The Coefficient of variation of delays

the TCP/IP delay is twice that of LightCommunicator For larger packet sizes the processing delays become predominant thus LightCommunicator performs considerably better than TCP/IP

Another observation from our experiment was that $8\ 4x10^8$ bytes were transacted in two directions using both the protocols No packets were lost in the transactions using LightCommunicator We cannot say anything about the losses when using TCP/IP

5 2 Summary and Future Work

We have developed LightCommunicator, a lightweight communication protocol and implemented in the Linux kernel The first step was to point those areas in TCP which are pure overheads in *friendly* situations Following that, we enumerated the desirable characteristics of a lightweight communication protocol for distributed computation in friendly environments, specified LightCommunicator and implemented it into the Linux kernel To the application layer LightCommunicator appears very similar to TCP/IP This means that we could modify the TCP/IP code to obtain the LightCommunicator

code Because of the similarity of the TCP/IP and LightCommunicator the modifications required in the application to use LightCommunicator for communication is minimal The only change that needs to be done is while creating a socket interface by calling the `socket()` system call Our experiments on the delay performance of LightCommunicator vis a vis TCP/IP show that the transaction delays can be reduced by more than half by using LightCommunicator instead of TCP/IP This advantage improves as the volume of data exchanged during a transaction increases

Our implementation is not yet efficient and some inefficiencies can be removed On the sending side, data copying has been reduced to one per message which is the best that can be done but on the receiving side we still have two instances of data copying per message This can be reduced further to one which will reduce the overhead further

There is also a possibility to implement *forwarding* This would be helpful if we want to use a topology other than a bus This will enable us to float multiple packets simultaneously on the multicomputer interconnects For example in a master-slave architecture a star topology may prove to be very advantageous Similarly toroids and grids may also be constructed Operating system support is also available for this purpose since Linux can support multiple Ethernet cards

Development of *net tools* for the protocol can also be done Currently all the information (about address and mappings) is stored in a file Tools may be written (similar to `ifconfig`) This would make its use easier

Bibliography

[1] David D Clark Van Jacobson John Romkey Howard Salwen An Analysis of TCP Processing Overhead 'IEEE *Communications Magazine* pp 23 29, June 1989

[2] Alan Cox Network buffer and memory management,' *Linux Journal* September 1996

[3] W Richard Stevens *UNIX Network Programming* Prentice Hall, 1994

[4] W Richard Stevens, *TCP/IP Illustrated, Vol 1* Addison Wesley Publishing Company 1994

[5] W Richard Stevens *TCP/IP Illustrated Vol 2* Addison Wesley Publishing Company 1994